NetLogo User's Guide

Programming Tutorial for synchronizing fireflies (adapted from the official tutorial)

NetLogo is a freeware program written in Java (it runs on all major platforms). You can download a version for your computer from
http://ccl.northwestern.edu/netlogo/

This tutorial was developped with Netlogo version 1.0, but the commands are equally valid for version 1.3 (a single difference is indicated in the text).

This tutorial describes a complete project, built up stage by stage, with every step explained along the way.

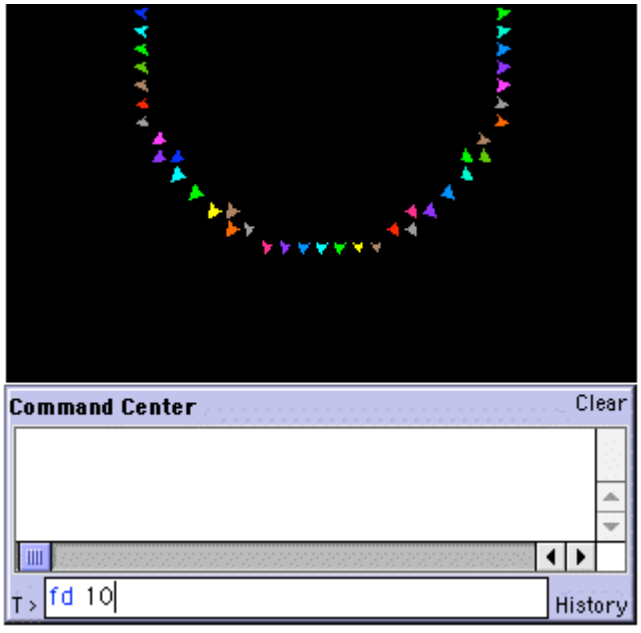
1. Creating a New Project: Getting Started

To create a new project, select "New" from the the File Menu. Let us first explore the Command Center which has a command line (as in Scilab, the other freeware program we will use in this module). We first create 100 turtles by the command

```
O> crt 100
```

NetLogo creates with this command 100 turtles (why turtles? well, that's the name the creators of NetLogo gave to their virtual individuals and we will adopt their terminology for the moment). But where are they? In fact you will only see one on the screen because one turtle sits on the other. To show them all we have to make them move away from the center. Since this only concerns the turtles we have to click on the O> in the command center and change it to T> (like **T**urtle), then type the command

```
T> fd 10
```

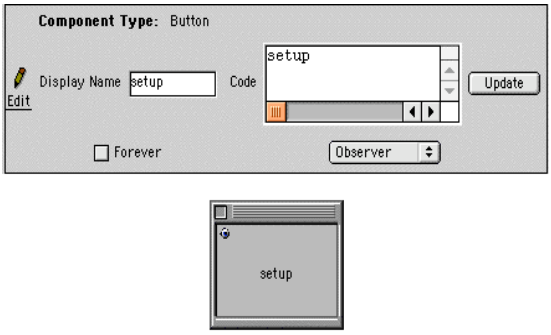


They now moved forward 10 steps. You can move them further away a random number of steps (between 0 and 9) with

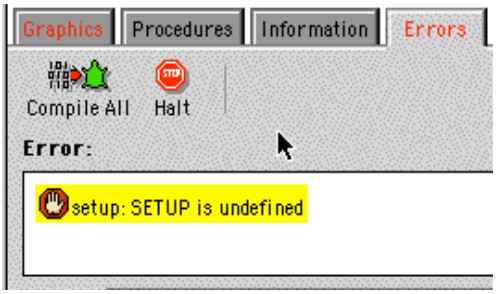
```
T> fd (random 10)
```

Finally, with the command **ca** (clear all, in the observer line O>) you will make disappear all the turtles.

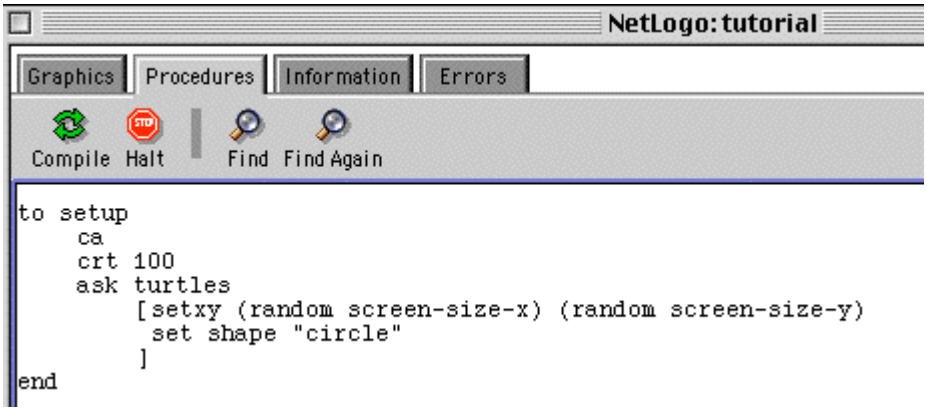
Typing all these commands on the command line permits to explore the functioning of NetLogo commands, but this is not very efficient if we want to repeat this sequence of commands several times. NetLogo permits to write commands also in a procedure. Start doing this by creating a once-button called 'setup'. This button will do just that -- initialize your project. You create this button by clicking on the Button icon on the Graphics Toolbar, and then clicking where you want the button to be in the Graphics Window. When the button appears, click on "edit" on the Toolbar, and type 'setup' in the code box. When clicking ok, a button will be created that is called 'setup' and will execute the procedure 'setup' when pressed.
In fact. you could name it something else, for example `Initialisation`. Try it out, move the cursor on the button, click the right mouse button and select the `Edit` menu: type `Initialisation` in the Display Name Window and click ok.



At this point, '`Errors`' will turn red. That's because there is no procedure called 'setup'!



Now go to the Procedures Window and create the 'setup' procedure like this:



One line at a time:

ca is short for **clear-all** (another acceptable NetLogo primitive that does the same thing). This command will blank out the screen, initialize any variables you might have to 0, and kill all turtles; basically it wipes the screen clean for a new run of the project.

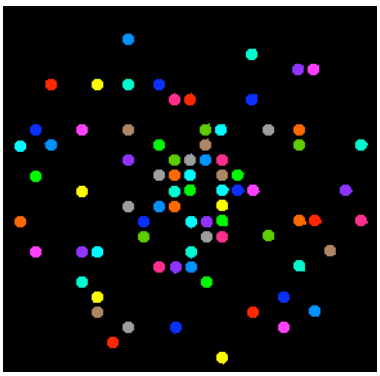
crt 100 will then create 100 turtles. Each of these turtles will begin at the origin (at location 0,0). That's why you only see what looks like one turtle; they're all on top of each other -- lots of turtles can share the same patch. Each of these newly-created turtles has its own color, evenly distributed through the range of NetLogo colors, and its own heading, evenly distributed around the circle.

ask turtles [ ] tells each turtle to execute, independently, the instructions inside the brackets . This is equivalent to selecting the command line T>... Note that **crt** is not in brackets. If the agent is not specified, the observer executes it. The third agent in NetLogo is 'patches' which are used to define a static environment. We won't use them in this tutorial.

Set shape "circle" sets all turtle shapes to one that you've created in the shapes editor (in the `Tools` in the menu bar). In this case, we made a shape that was a simple circle, then named it "circle". If you don't make a shape, the turtles will have a default shape.

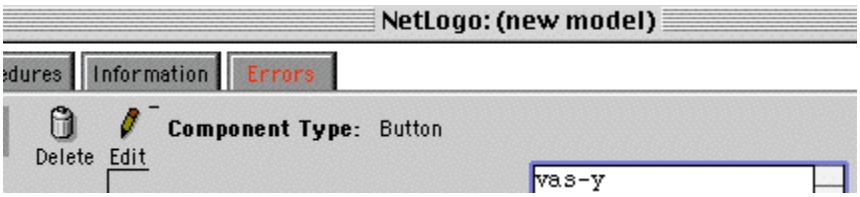
Finally, **setxy (random screen-size-x) (random screen-size-y** is really a pair of commands. Each turtle will first evaluate the commands **random screen-edge-x** and **random screen-size-y** which will return each a random value between 0 and 'screen-edge' (the dimension from the center to the edge of the screen along the x-axis or y-axis, maybe 20 or so). It then takes this pair of numbers, and sets the turtles at these random coordinates.

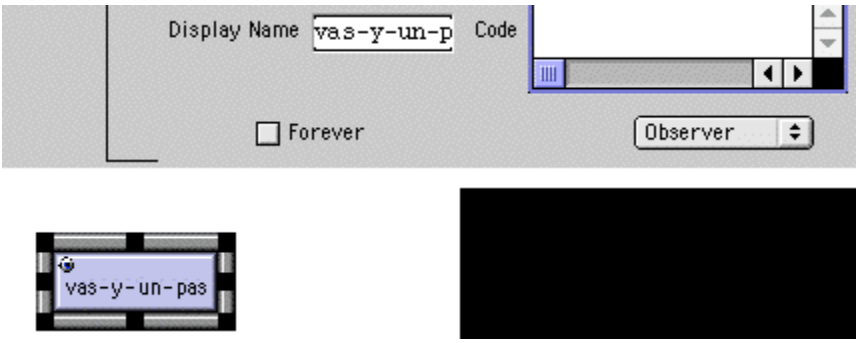
Press your 'setup' button when you've written the code. You can see the turtles quickly spread out in a rough cluster.



Notice the density distribution of the turtles on the Graphics Screen. Press 'setup' a couple more times, and watch how the turtles' arrangement changes. Can you think of other ways to randomly distribute the turtles over the screen? (There are lots of ways.) Note that if a turtle moves off the screen, it "wraps", that is, comes in the other side. When you're satisfied that the button does what you want, it's time to write your next procedure. But first you should save your work (in order to preserve it in case of a computer crash): create a folder with your name on the harddisk C and then use the menu `File -> Save As ...` to save your project in this folder. During this tutorial it will be a good idea to save your work from time to time with the menu `File -> Save`.

Now to the next procedure: make first a button called 'vas-y-un-pas'. Again, begin by creating a button with the Display Name 'vas-y-un-pas' and the code command 'vas-y'.





Then write the procedure

```
to vas-y
    bouge
end
```

But what is **bouge** ? Is it a primitive, like **fd** is? No, it's a procedure that you're about to write, right after the **vas-y** procedure:

```
to bouge
  ask turtles [
    set heading random 360
    fd 1
  ]
end
```

Did you notice anything unusual like a change to smaller characters? Bad luck, you have a french keyboard, and to type **]** you have to press `Alt-)` which is interpreted by NetLogo as a shortcut to reduce letter size. You can revert the change with the menu `Zoom-Normal Size` (or `Ctrl-=`) and avoid this bug in the future by copying and pasting **]**.

Now, look at the procedure line by line:

ask turtles [ ] says that each turtle should execute the commands in the brackets.

set heading random 360 is another two-step command. First, each turtle picks a random integer between 0 and 359 (**random** doesn't include the number you give it). Then it sets its heading to be this number. Heading is measured in degrees, clockwise around the circle, starting with 0 degrees at twelve o'clock.

fd 1: After each turtle does that, it moves forward one step in this new direction.

Why couldn't we have just written that in **vas-y** ? We could, but during the course of building your project, it's likely that you'll add many other parts. We'd like to keep **vas-y** as simple as possible, so that it is easy to understand. It could include many other things you want to have happen as the model runs, such as calculating something or plotting the results. Each of these sub-procedures could have its own name.

Each time you click now on **vas-y** NetLogo will execute the commands in **bouge**. But it would be nice if NetLogo would repeat **bouge** indefinitely until you click again on the button. This can be done easily : create another button with the Name 'vas-y' and the code **vas-y**, but this time check the "forever" box in the editing dialog box.

The 'vas-y' button is a forever-button, meaning that it will continually execute its code until you shut it off (by clicking on it again). After you have pressed 'setup' once, to create the burls, press the 'vas-y' button. Watch what happens. Turn it off, and you'll see that all turtles stop in their tracks.

You could probably jump off from here and start experimenting with other turtle commands. That

isn't such a bad idea. Type commands into the Command Window (like **set color** *red*), or add them to ***setup***, ***vas-y***, or ***vas-y-un-pas***. Note that when you enter commands in the Command Center, you must choose **T>**, **P>**, or **O>** in the drag-down menu on the left, depending on which agents are going to execute the commands. **T>*commands*** is identical to **O> ask turtles [*commands*]**, and **P>*commands*** is identical to **O> ask patches [*commands*]** (patches are a way to incorporate a dynamic environment, e.g. grains transported by ants, but we won't use them in this tutorial).

You might try typing **T> lt** (**random 45**) (for "left turn", instead of T> **set heading** (**random 360**)), or any other NetLogo command that applies to turtles or patches. But first, we will make the number of turtles a user-defined variable. Instead of saying **crt 100** in **setup**, say **crt** *nombre*, where *nombre* is a slider variable, and create a slider from the graphics toolbar just as you created the buttons (with minimum value 0 and maximum value 2000 in steps of 10). When you change now the number of turtles with the slider and click **Setup** again NetLogo will display the new number of Turtles. Play around. It's easy and the results are immediate and visible -- one of NetLogo's many strengths. Regardless, the tutorial project continues...

2. Developing Your Model: From Turtles to Fireflies with their variables

So now we've got 100 turtles aimlessly moving around, completely unaware of anything else around them. Actually, we want our turtles to be fireflies and to behave like them, that is to synchronize their flashing when they meet other flashing fireflies. For this we have to give our fireflies some variables: **compteur** which is the internal clock of the firefly (it will flash when its value is 0 and increase at each step to a maximum of 9 after which it is set back to 0), **seuil** which is the value of the **compteur** at which the firefly stops flashing, **niveau** which is the level the **compteur** is set to when seeing a flashing firefly, and **fenetre** which is an insensitivity period after flashing during which the firefly doesn't see any other fireflies. Well, let's do all this step by step. First, add the variables at the top of the procedures window.

```
turtles-own
[ compteur ; horloge de chaque luciole, va de 0 a 9 auquel valeur elle
           ; clignote
  seuil    ; seuil du compteur auquel la luciole arrete le clignotement
  niveau   ; niveau auquel une luciole met son compteur quand elle voit
           ; un voisin clignoter
  fenetre  ; une luciole ne peut changer son compteur s'il est <= fenetre
]
```

The semicolon indicates that the rest of the line contains comments that are ignored by NetLogo and help a reader to understand the code, but you do not have to include these comments in your current code. We now have to set this variables in **setup** to reasonable values, e.g. fireflies should be yellow when flashing and gray otherwise.

```
to setup
     ca
        crt nombre
        ask turtles
                [setxy (random screen-size-x) (random screen-size-y)
                 set shape "circle"
                 set compteur random (round 10) ;; 10 est la duree d'un cycle
                 set seuil 1
                 set niveau seuil
                 set fenetre -1 ;; pas de periode insensible
                 ifelse (compteur < seuil)
                        [ set color yellow ]
                        [ set color gray ]
                ]
end
```

We used here a new NetLogo command, **ifelse (test) [commands1] [commands2],** that executes **commands1** if **test** is T (true), **commands2** otherwise. Note also that it is quite unnatural for

fireflies to turn around completely randomly in the **bouge** routine, they rather do a random left wiggle and a random right wiggle before continuing their path. The new **bouge** routine thus becomes

```
to bouge
        ask turtles [
                set heading heading + random 90 - random 90
                fd 1
        ]
end
```

In **vas-y** we have to add a routine that increases **compteur** at each step and resets the firefly color according to the value of **compteur**.

```
to vas-y
        bouge
        ask turtles [
                augmente-compteur
                ifelse (compteur < seuil)
                [ set color yellow ]
                [ set color gray ]
        ]
end

to augmente-compteur
        set compteur (compteur + 1)
        if compteur = 10
        [ set compteur 0 ]
end
```

We use here the new command **if (test) [commands]** that is a simplified version of **ifelse.** If you go back to your interface window and click on `vas-y-un-pas` you will see that yellow turtles become gray on the next step.

3. Adding local vision and synchronization

Our fireflies move now around in a rather natural manner and flash every 10 steps. Why are they not synchronizing? Well, simply because we still haven't implemented the mechanism that is supposed to lead to synchronisation: when a neighbouring firefly flashes, then the **compteur** is set back to **seuil**. We therefore need a routine **regarde** that detects flashing fireflies in the neighbourhood.

```
to regarde
        if ( count turtles in-radius 1 with [color = yellow] >= 1 )
                [set compteur niveau ]
end
```

Do you understand what the test does? It first counts the flashing fireflies in a Moore neighbourhood and then checks if this number is larger than one. This routine is called after **augmente-compteur** if the firefly is not flashing itself and is again sensitive for other flashes.

```
to vas-y
 bouge
        ask turtles [
                augmente-compteur
                if ( (compteur > fenetre) and (compteur >= seuil) )
                [ regarde ]
                ifelse (compteur < seuil)
                [ set color yellow ]
                [ set color gray ]
        ]
end
```

Thats it, the mechanism that should synchronize flashing is now implemented. Does it work?

4. Adding observer windows and visualization of the synchronizing process

We now have a functioning model of synchronizing fireflies. It can be used to study the influence of the different environmetal variables on the synchronization process. How long does it take to synchronize 80% of the fireflies? How does the number of fireflies influence this time? How does the value of **fenetre** influence this time? This playing around with the model helps to detect new patterns and to formulate new hypotheses that can then be tested under natural conditions.

To do so we first need to add a global variable **ticks** that keeps track of the number of steps our simulation has run. Add the line

```
globals [ ticks ]
```

after the **turtles-own [...]** command. Add a command in the **setup** routine to set it to 0 (you should now know how to do this) and increase it by 1 in the **vas-y** routine. Then we should display this number on the screen. Have you already seen the button we need to do that? Exactly, we will *monitor* the ticks, so simply click on the **Monitor** button and create it in the Graphics window, telling it to display the value of **ticks**. Now, you should see 0 in this monitor after **setup** and it should increase by one each time you click on **vas-y-un-pas**.

Similarly, it would be nice to know how many fireflies are flashing at each time step. For this we need a second global variable **allume** (this should be **allumé**, but using accents in english software is always playing with the devil, most english programmers don't even know accents exist and thus have often simply forgotten to tell the program what to do with such strange characters) that is displayed in its own monitor and a routine that counts the number of flashing fireflies.

```
globals [ ticks
                allume ]

to compte-allume
        set allume (count turtles with [color = yellow] )
end
```

compte-allume can be called at the end of the **vas-y** routine, and the monitor for **allume** is created as the one for **Ticks**. Simulating now step by step permits to determine exactly at which step more than 80% of the fireflies are flashing synchronously. Try to do it.

Got it? Takes too much time? I agree completely. It would be nice to have a graph that plots the number of **allume** as a function of **Ticks**. Again, NetLogo provides the tools to do this. We first need routines that initialize and update this graph during run-time. In the **setup** procedure, add a call to the routine **creer-graph**

```
to creer-graph
        set-current-plot "Lucioles allumees"
        auto-plot-on
        set-plot-x-range 0 100
        set-plot-y-range 0 nombre
        set-plot-pen "lucioles" ; this line is obsolete in 1.3
        set-plot-pen-color red
end
```

and in the **vas-y** routine, add a call to **faire-graph**

```
to faire-graph
        set-current-plot "Lucioles allumees"
        plot allume
end
```

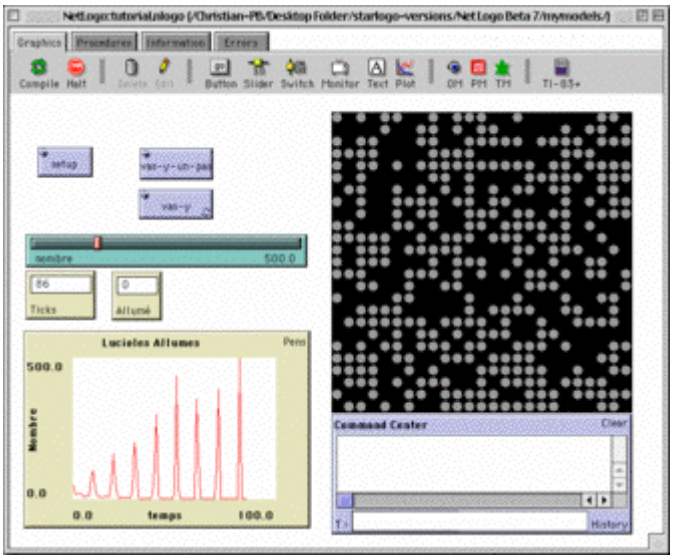
What we're plotting is the number of flashing fireflies at some given time. Step by step, the commands in our **creer-graph** procedure do the following:

- Select the first plot window
- Turn on automatic rescaling of the plot window axes (useful if you're plotting some value over time, as we are)
- Set the range of the x-axis, both minimum (0) and maximum (100)
- Set the range of the y-axis, again both minimum and maximum (because we'll have, at most, **nombre** fireflies at peaks, our maximum is the number of fireflies)
- Name the first plot pen to be the current active plot pen in this graph
- Set the plot pen color to be red

Then in **faire-graph**, we select the plot "Lucioles allumes" and the plot pen "lucioles" (currently redundant, since there's just one , but maybe we'll add more later...), and then we plot a point.

The **plot** command draws a connected graph at the current x-coordinate and y-coordinate and then updates the x-coordinate by 1 automatically. But where will it draw? Go back to the "Interface" window, there you can add a plot widget to your model, edit it and change its name to "In order for `set-current-plot "Lucioles allumees"Lucioles allumees`", the exact same name used in the code.

Now reset the project and run it again. Watch the graph in the Plot Window. You might try running the model several times under different settings (i.e. different values of **nombre**) and watch how fast the graph reaches 80%.



So now you have a nice framework for exploring the fireflies synchronization problem, using all sorts of NetLogo modeling features: buttons, sliders, monitors, and the graphics and plot window displays. You've even written a quick and dirty procedure to give the turtles something to do. And that's where this tutorial leaves off. You can continue it if you'd like, experimenting with different variables and algorithms to see what works the best (what makes the most turtles reach the peaks). Alternatively you can look at other models, or go ahead and build your own. You don't even have to model anything. It can be pleasant just to watch patches and turtles forming patterns, or whatever. Hopefully you will have learned a few things, both in terms of syntax and general methodology for model-building.

5. Homework (the bad news ;-)

As a homework, use your program to explore how the number of fireflies influences the time it takes to synchronize at least 80% of the fireflies (set **nombre** to 10, 20, 50, 100, 200, 500, 1000) and get a confidence interval of this duration by repeating each simulation a couple of times. With a fixed number of fireflies (e.g. 500), guess intuitively how the value of **fenetre** influences this time.

Modify your program to explore whether your intuition gave you the correct answer (use the values 0, 1, 2, and 3 for **fenetre**). How do you interpret this effect biologically? Any other intuitions? Write a nice little report that summarizes your results and interpretations (1-3 pages). It may be in French.

ps: for more information on NetLogo and the detailed definitions of NetLogo commands you can view the documentation provided with your installation of NetLogo. Start your web browser (Netscape, Explorer, ...) and open the local file `.../docs/index.html` (in your Netlogo Folder) with the menu "File->Open File ..." (or suchlike).
You can also visit directly the NetLogo internet site,
http://ccl.northwestern.edu/netlogo/
it has lot's of useful information.

16 octobre 2003, Christian Jost.